Chapter 4: Driver from Start to Finish

In this chapter, we'll use many of the concepts we learned in previous chapters and build a simple, yet complete driver and client application, while filling in some of the missing details. We'll deploy the driver and use its capabilities - perform some operation in kernel mode that is not available to user mode.

In this chapter:

- **Introduction**
- **Driver Initialization**
- **Client Code**
- **The Create and Close Dispatch Routines**
- **The `DeviceIoControl` Dispatch Routine**
- **Installing and Testing**

Introduction

The problem we'll solve with a simple kernel driver is the inflexibility of setting thread priorities using the Windows API. In user mode, a thread's priority is determined by a combination of its process *Priority Class* with an offset on a per thread basis that has a limited number of levels. Changing a process priority class can be achieved with the `SetPriorityClass` function that accepts a process handle and one of the six supported priority classes. Each priority class corresponds to a priority level, which is the default priority of threads created in that process. A particular thread's priority can be changed with the `SetThreadPriority` function, accepting a thread handle and one of several constants corresponding to offsets around the base priority class. Table 4-1 shows the available thread priorities based on the process priority class and the thread's priority offset.

Table 4-1: Legal values for thread priorities with the Windows APIs

Priority Class	- Sat	-2	-1	0 (default)	+1	+2	+ Sat	Comments
Idle (Low)	1	2	3	4	5	6	15	
Below Normal	1	4	5	6	7	8	15	
Normal	1	6	7	8	9	10	15	
Above Normal	1	8	9	10	11	12	15	
High	1	11	12	13	14	15	15	Only six levels are available (not seven).
Real-time	16	22	23	24	25	26	31	All levels between 16 to 31 can be selected.

The values acceptable to `SetThreadPriority` specify the offset. Five levels correspond to the offsets -2 to +2: `THREAD_PRIORITY_LOWEST` (-2), `THREAD_PRIORITY_BELOW_NORMAL` (-1), `THREAD_PRIORITY_NORMAL` (0), `THREAD_PRIORITY_ABOVE_NORMAL` (+1), `THREAD_PRIORITY_HIGHEST` (+2). The remaining two levels, called *Saturation* levels, set the priority to the two extremes supported by that priority class: `THREAD_PRIORITY_IDLE` (-Sat) and `THREAD_PRIORITY_TIME_CRITICAL` (+Sat).

The following code example changes the current thread's priority to 11:

```
SetPriorityClass(GetCurrentProcess(), ABOVE_NORMAL_PRIORITY_CLASS);
SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_ABOVE_NORMAL);
```

The Real-time priority class does not imply Windows is a real-time OS; Windows does not provide some of the timing guarantees normally provided by true real-time operating systems. Also, since Real-time priorities are very high and compete with many kernel threads doing important work, such a process must be running with administrator privileges; otherwise, attempting to set the priority class to Real-time causes the value to be set to High.

There are other differences between the real-time priorities and the lower priority classes. Consult the *Windows Internals* book for more information.

Table 4-1 shows the problem we will address quite clearly. Only a small set of priorities are available to set directly. We would like to create a driver that would circumvent these limitations and allow setting a thread's priority to any number, regardless of its process priority class.

Driver Initialization

We'll start building the driver in the same way we did in chapter 2. Create a new "WDM Empty Project" named *PriorityBooster* (or another name of your choosing) and delete the INF file created by the wizard. Next, add a new source file to the project, called *PriorityBooster.cpp* (or any other name you prefer). Add the basic #include for the main WDK header and an empty `DriverEntry`:

```
#include <ntddk.h>

extern "C" NTSTATUS
DriverEntry(_In_ PDRIVER_OBJECT DriverObject, _In_ PUNICODE_STRING RegistryPath) {
    return STATUS_SUCCESS;
}
```

Most software drivers need to do the following in `DriverEntry`:

- Set an Unload routine.
- Set dispatch routines the driver supports.
- Create a device object.
- Create a symbolic link to the device object.

Once all these operations are performed, the driver is ready to take requests.

The first step is to add an Unload routine and point to it from the driver object. Here is the new `DriverEntry` with the Unload routine:

```
// prototypes

void PriorityBoosterUnload(_In_ PDRIVER_OBJECT DriverObject);

// DriverEntry

extern "C" NTSTATUS
DriverEntry(_In_ PDRIVER_OBJECT DriverObject, _In_ PUNICODE_STRING RegistryPath) {
    DriverObject->DriverUnload = PriorityBoosterUnload;

    return STATUS_SUCCESS;
}

void PriorityBoosterUnload(_In_ PDRIVER_OBJECT DriverObject) {
}
```

We'll add code to the Unload routine as needed when we do actual work in `DriverEntry` that needs to be undone.

Next, we need to set up the dispatch routines that we want to support. Practically all drivers must support `IRP_MJ_CREATE` and `IRP_MJ_CLOSE`, otherwise there would be no way to open a handle to any device for this driver. So we add the following to `DriverEntry`:

```
    DriverObject->MajorFunction[IRP_MJ_CREATE] = PriorityBoosterCreateClose;
    DriverObject->MajorFunction[IRP_MJ_CLOSE] = PriorityBoosterCreateClose;
```

We're pointing the Create and Close major functions to the same routine. This is because, as we'll see shortly, they will actually do the same thing: simply approve the request. In more complex cases, these could be separate functions, where in the Create case the driver can (for instance) check to see who the caller is and only let approved callers succeed with opening a device.

All major functions have the same prototype (they are part of an array of function pointers), so we have to add a prototype for `PriorityBoosterCreateClose`. The prototype for these functions is as follows:

```
NTSTATUS PriorityBoosterCreateClose(_In_ PDEVICE_OBJECT DeviceObject, _In_ PIRP Irp);
```

The function must return `NTSTATUS` and accepts a pointer to a device object and a pointer to an *I/O Request Packet* (IRP). An IRP is the primary object where the request information is stored, for all types of requests. We'll dig deeper into an IRP in chapter 6, but we'll look at the basics later in this chapter, since we require it to complete our driver.

Passing Information to the Driver

The Create and Close operations we set up are required, but certainly not enough. We need a way to tell the driver which thread and to what value to set its priority. From a user mode client's perspective, there are three basic functions it can use: `WriteFile`, `ReadFile` and `DeviceIoControl`.

For our driver's purposes, we can use either `WriteFile` or `DeviceIoControl`. Read doesn't make sense, because we're passing information *to* the driver, rather than from the driver. So which is better, `WriteFile` or `DeviceIoControl`? This is mostly a matter of taste, but the general wisdom here is to use Write if it's really a write operation (logically); for anything else - `DeviceIoControl` is preferred, as it's a generic mechanism for passing data to and from the driver.

Since changing a thread's priority is not a purely Write operation, we'll go with `DeviceIoControl`. This function has the following prototype:

```
BOOL WINAPI DeviceIoControl(
    _In_ HANDLE hDevice,
    _In_ DWORD dwIoControlCode,
    _In_reads_bytes_opt_(nInBufferSize) LPVOID lpInBuffer,
    _In_ DWORD nInBufferSize,
    _Out_writes_bytes_to_opt_(nOutBufferSize,*lpBytesReturned) LPVOID lpOutBuffer,
    _In_ DWORD nOutBufferSize,
    _Out_opt_ LPDWORD lpBytesReturned,
    _Inout_opt_ LPOVERLAPPED lpOverlapped);
```

There are three important pieces to `DeviceIoControl`:

- A control code
- An input buffer
- An output buffer

This means `DeviceIoControl` is a flexible way to communicate with a driver. Several control codes can be supported which would require different semantics passed along with the optional buffers. On the driver side, `DeviceIoControl` corresponds to the `IRP_MJ_DEVICE_CONTROL` major function code. Let's add it to our initialization of dispatch routines:

```
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = PriorityBoosterDeviceControl;
```

Client / Driver Communication Protocol

Given that we decided to use `DeviceIoControl` for client/driver communication, we now must define the actual semantics. Clearly, we need a control code and the input buffer. This buffer should contain the two pieces of information required so the driver can do its thing: the thread id and the priority to set for it.

These pieces of information must be usable both by the driver and the client. The client would supply the data, and the driver would act on it. This means these definitions must be in a separate file that must be included by the driver and client code.

For these purpose, we'll add a header file named *PriorityBoosterCommon.h* to the driver project. This file will also be used later by the user mode client.

Within this file, we need to define two things: the data structure the driver expects from clients and the control code for changing a thread's priority. Let's start by declaring a structure that captures the information the driver needs for a client:

```
struct ThreadData {
    ULONG ThreadId;
    int Priority;
};
```

We need the thread's unique ID and the target priority. Thread IDs are 32-bit unsigned integers, so we select `ULONG` as the type (Note that we cannot normally use `DWORD` - a common type defined in user mode headers - because it's not defined in kernel mode headers. `ULONG`, on the other hand, is defined in both). The priority should be a number between 1 and 31, so a simple 32-bit integer will do.

Next we need to define a control code. You may think that any 32-bit number will do, but that is not the case. The control code must be built using the `CTL_CODE` macro, that accepts four arguments that make up the final control code. `CTL_CODE` is defined like so:

```
#define CTL_CODE( DeviceType, Function, Method, Access ) ( \
    ((DeviceType) << 16) | ((Access) << 14) | ((Function) << 2) | (Method))
```

Here is a brief description of the meaning of these macro arguments:

- *DeviceType* - identifies a type of device. This can be one of the `FILE_DEVICE_xxx` constants defined in the WDK headers, but this is mostly for hardware based drivers. For software drivers like ours, the number doesn't matter much. Still, Microsoft's documentation specifies that values for 3rd parties should start with `0x8000`.
- *Function* - an ascending number indicating a specific operation. If nothing else, this number must be different between different control codes for the same driver. Again, any number will do, but the official documentation says 3rd party drivers should start with `0x800`.
- *Method* - the most important part of the control code. It indicates how the input and output buffers provided by the client pass to the driver. We'll deal with these values in detail in chapter 6. For our driver, we'll use the simplest value `METHOD_NEITHER`. We'll see its effect later in this chapter.
- *Access* - indicates whether this operation is to the driver (`FILE_WRITE_ACCESS`), from the driver (`FILE_READ_ACCESS`), or both ways (`FILE_ANY_ACCESS`). Typical drivers just use `FILE_ANY_ACCESS` and deal with the actual request in the `IRP_MJ_DEVICE_CONTROL` handler.

Given the above information, we can define our single control code as follows:

```
#define PRIORITY_BOOSTER_DEVICE 0x8000

#define IOCTL_PRIORITY_BOOSTER_SET_PRIORITY CTL_CODE(PRIORITY_BOOSTER_DEVICE, \
    0x800, METHOD_NEITHER, FILE_ANY_ACCESS)
```

Creating the Device Object

We have more initializations to do in `DriverEntry`. Currently, we don't have any device object and so there is no way to open a handle and reach the driver. A typical software driver needs just one device object, with a symbolic link pointing to it, so that user mode clients can obtain handles.

Creating the device object requires calling the `IoCreateDevice` API, declared as follows (some SAL annotations omitted/simplified for clarity):

(C)2019 Pavel Yosifovich

```
NTSTATUS IoCreateDevice(
    _In_        PDRIVER_OBJECT DriverObject,
    _In_        ULONG DeviceExtensionSize,
    _In_opt_    PUNICODE_STRING DeviceName,
    _In_        DEVICE_TYPE DeviceType,
    _In_        ULONG DeviceCharacteristics,
    _In_        BOOLEAN Exclusive,
    _Outptr_    PDEVICE_OBJECT *DeviceObject);
```

The arguments to `IoCreateDevice` are described below:

- *DriverObject* - the driver object to which this device object belongs to. This should be simply the driver object passed to the `DriverEntry` function.
- *DeviceExtensionSize* - extra bytes that would be allocated in addition to `sizeof(DEVICE_OBJECT)`. Useful for associating some data structure with a device. It's less useful for software drivers creating just a single device object, since the state needed for the device can simply be managed by global variables.
- *DeviceName* - the internal device name, typically created under the *Device* Object Manager directory.
- *DeviceType* - relevant to some type of hardware based drivers. For software drivers the value `FILE_DEVICE_UNKNOWN` should be used.
- *DeviceCharacteristics* - a set of flags, relevant for some specific drivers. Software drivers specify zero or `FILE_DEVICE_SECURE_OPEN` if they support a true namespace (rarely used by software drivers, beyond the scope of this book).
- *Exclusive* - should more than one file object be allowed to open the same device? Most drivers should specify `FALSE`, but in some cases `TRUE` is more appropriate; it forces a single client to the device.
- *DeviceObject* - the returned pointer, passed as pointer to a pointer. If successful, `IoCreateDevice` allocates the structure from non-paged pool and stores the resulting pointer inside the dereferenced argument.

Before calling `IoCreateDevice` we must create a `UNICODE_STRING` to hold the internal device name:

```
UNICODE_STRING devName = RTL_CONSTANT_STRING(L"\\Device\\PriorityBooster");
// RtlInitUnicodeString(&devName, L"\\Device\\ThreadBoost");
```

The device name could be anything but should be in the *Device* object manager directory. There are two ways to initialize a `UNICODE_STRING` with a constant string. The first is using `RtlInitUnicodeString`, which works just fine. But `RtlInitUnicodeString` must count the number of characters in the string to initialize the `Length` and `MaximumLength` appropriately. Not a big deal in this case, but there is a quicker way - using the `RTL_CONSTANT_STRING` macro, which

calculates the length of the string statically at compile time, meaning it can only work correctly with constant strings.

Now we can call the `IoCreateDevice` function:

```
PDEVICE_OBJECT DeviceObject;
NTSTATUS status = IoCreateDevice(
    DriverObject        // our driver object,
    0                   // no need for extra bytes,
    &devName            // the device name,
    FILE_DEVICE_UNKNOWN // device type,
    0                   // characteristics flags,
    FALSE               // not exclusive,
    &DeviceObject       // the resulting pointer
    );
if (!NT_SUCCESS(status)) {
    KdPrint(("Failed to create device object (0x%08X)\n", status));
    return status;
}
```

If all goes well, we now have a pointer to our device object. The next step is make this device object accessible to user mode callers by providing a symbolic link. The following lines create a symbolic link and connect it to our device object:

```
UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\\??\\PriorityBooster");
status = IoCreateSymbolicLink(&symLink, &devName);
if (!NT_SUCCESS(status)) {
    KdPrint(("Failed to create symbolic link (0x%08X)\n", status));
    IoDeleteDevice(DeviceObject);
    return status;
}
```

The `IoCreateSymbolicLink` does the work by accepting the symbolic link and the target of the link. Note that if the creation fails, we must undo everything done so far - in this case just the fact the device object was created - by calling `IoDeleteDevice`. More generally, if `DriverEntry` returns any failure status, the Unload routine is **not** called. If we had more initialization steps to do, we would have to remember to undo everything until that point in case of failure. We'll see a more elegant way of handling this in chapter 5.

Once we have the symbolic link and the device object set up, `DriverEntry` can return success and the driver is now ready to accept requests.

Before we move on we must not forget the Unload routine. Assuming `DriverEntry` completed successfully, the Unload routine must undo whatever was done in `DriverEntry`. In our case, there are two things: device object creation and symbolic link creation. We'll undo them in reverse order:

```
void PriorityBoosterUnload(_In_ PDRIVER_OBJECT DriverObject) {
    UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\\??\\PriorityBooster");
    // delete symbolic link
    IoDeleteSymbolicLink(&symLink);

    // delete device object
    IoDeleteDevice(DriverObject->DeviceObject);
}
```

Client Code

At this point it's worth writing the user mode client code. Everything we need for the client has already been defined.

Add a new console desktop project to the solution named *Booster* (or some other name of your choosing). The Visual Studio wizard should create a single source file (Visual Studio 2019), and two precompiled header files (*pch.h*, *pch.cpp*) in Visual Studio 2017. You can safely ignore the precompiled header files for now.

In the *Booster.cpp* file, remove the default "hello, world" code and add the following declaration:

```
#include <windows.h>
#include <stdio.h>
#include "..\PriorityBooster\PriorityBoosterCommon.h"
```

Note that we include the common header file created by the driver and shared with client code.

Change the main function to accept command line arguments. We'll accept a thread ID and a priority using command line arguments and request the driver to change the priority of the thread to the given value.

```
int main(int argc, const char* argv[]) {
    if (argc < 3) {
        printf("Usage: Booster <threadid> <priority>\n");
        return 0;
    }
```

Next we need to open a handle to our device. The "file name" to `CreateFile` should be the symbolic link prepended with "\\.\". The entire call should look like this:

```
HANDLE hDevice = CreateFile(L"\\\\.\\PriorityBooster", GENERIC_WRITE,
    FILE_SHARE_WRITE, nullptr, OPEN_EXISTING, 0, nullptr);
if (hDevice == INVALID_HANDLE_VALUE)
    return Error("Failed to open device");
```

The *Error* function simply prints some text with the last error occurred:

```
int Error(const char* message) {
    printf("%s (error=%d)\n", message, GetLastError());
    return 1;
}
```

The `CreateFile` call should reach the driver in its `IRP_MJ_CREATE` dispatch routine. If the driver is not loaded at this time - meaning there is no device object and no symbolic link - we'll get an error number 2 (file not found). Now that we have a valid handle to our device, it's time to set up the call to `DeviceIoControl`. First, we need to create a `ThreadData` structure and fill in the details:

```
ThreadData data;
data.ThreadId = atoi(argv[1]);  // command line first argument
data.Priority = atoi(argv[2]);  // command line second argument
```

Now we're ready to call `DeviceIoControl` and close the device handle afterwards:

```
DWORD returned;
BOOL success = DeviceIoControl(hDevice,
    IOCTL_PRIORITY_BOOSTER_SET_PRIORITY,    // control code
    &data, sizeof(data),                    // input buffer and length
    nullptr, 0,                             // output buffer and length
    &returned, nullptr);
if (success)
    printf("Priority change succeeded!\n");
else
    Error("Priority change failed!");

CloseHandle(hDevice);
```

`DeviceIoControl` reaches the driver by invoking the `IRP_MJ_DEVICE_CONTROL` major function routine.

At this point the client code is done. All that remains is to implement the dispatch routines we declared on the driver side.

The Create and Close Dispatch Routines

Now we're ready to implement the three dispatch routines defined by the driver. The simplest by far are the Create and Close routines. All that's needed is completing the request with a successful status. Here is the complete Create/Close dispatch routine implementation:

```
_Use_decl_annotations_
NTSTATUS PriorityBoosterCreateClose(PDEVICE_OBJECT DeviceObject, PIRP Irp) {
    UNREFERENCED_PARAMETER(DeviceObject);

    Irp->IoStatus.Status = STATUS_SUCCESS;
    Irp->IoStatus.Information = 0;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return STATUS_SUCCESS;
}
```

Every dispatch routine accepts the target device object and an *I/O Request Packet* (IRP). We don't care much about the device object, since we only have one, so it must be the one we created in `DriverEntry`. The IRP on the other hand, is extremely important. We'll dig deeper into IRPs in chapter 6, but we need to take a quick look at IRPs now.

An IRP is a semi-documented structure that represents a request, typically coming from one of the managers in the Executive: I/O Manager, Plug & Play Manager or Power Manager. With a simple software driver, that would most likely be the I/O Manager. Regardless of the creator of the IRP, the driver's purpose is to handle the IRP, which means looking at the details of the request and doing what needs to be done to complete it.

Every request to the driver always arrives wrapped in an IRP, whether that's a Create, Close, Read or any other IRP. By looking at the IRP's members, we can figure out the type and details of the request (technically, the dispatch routine itself was pointed to based on the request type, so in most cases you already know the request type). It's worth mentioning that an IRP never arrives alone; it's accompanied by one or more structures of type `IO_STACK_LOCATION`. In simple cases like our driver, there is a single `IO_STACK_LOCATION`. In more complex cases where there are filter drivers above or below us, multiple `IO_STACK_LOCATION` instances exist, one for each layer in the device stack. (We'll discuss this more thoroughly in chapter 6). Simply put, some of the information we need is in the base `IRP` structure and some is in the `IO_STACK_LOCATION` for our "level" in the device stack.

In the case of Create and Close, we don't need to look into any members. We just need to set the status of the IRP in its `IoStatus` member (of type `IO_STATUS_BLOCK`), which has two members:

- *Status* - indicating the status this requests would complete with.
- *Information* - a polymorphic member, meaning different things in different requests. In the case of Create and Close, a zero value is just fine.

(C)2019 Pavel Yosifovich

To actually complete the IRP, we call `IoCompleteRequest`. This function has a lot to do, but basically it propagates the IRP back to its creator (typically the I/O Manager) and that manager notifies the client that the operation has completed. The second argument is a temporary priority boost value that a driver can provide to its client. In most cases a value of zero is best (`IO_NO_INCREMENT` is defined as zero), because the request completed synchronously, so no reason the caller should get a priority boost. Again, more information on this function is provided in chapter 6.

The last operation to do is return the same status as the one put into the IRP. This may seem like a useless duplication, but it is necessary (the reason will be clearer in a later chapter).

The `DeviceIoControl` Dispatch Routine

This is the crux of the matter. All the driver code so far has lead to this dispatch routine. This is the one doing the actual work of setting a given thread to a requested priority.

The first thing we need to check is the control code. Typical drivers may support many control codes, so we want to fail the request immediately if the control code is not recognized:

```
_Use_decl_annotations_
NTSTATUS PriorityBoosterDeviceControl(PDEVICE_OBJECT, PIRP Irp) {
    // get our IO_STACK_LOCATION
    auto stack = IoGetCurrentIrpStackLocation(Irp); // IO_STACK_LOCATION*
    auto status = STATUS_SUCCESS;

    switch (stack->Parameters.DeviceIoControl.IoControlCode) {
        case IOCTL_PRIORITY_BOOSTER_SET_PRIORITY:
            // do the work
            break;

        default:
            status = STATUS_INVALID_DEVICE_REQUEST;
            break;
    }
```

The key to getting the information for any IRP is to look inside the `IO_STACK_LOCATION` associated with the current device layer. Calling `IoGetCurrentIrpStackLocation` returns a pointer to the correct `IO_STACK_LOCATION`. In our case, there is really just one `IO_STACK_LOCATION`, but in any case calling `IoGetCurrentIrpStackLocation` is the right call to make.

The main ingredient in an `IO_STACK_LOCATION` is a monstrous union member named `Parameters` which holds a set of structures, one for each type of IRP. In the case of `IRP_MJ_DEVICE_CONTROL` the structure to look at is `DeviceIoControl`. In that structure we can find the information conveyed by the client, such as the control code, the buffers and their lengths.

(C)2019 Pavel Yosifovich

The `switch` statement uses the `IoControlCode` member to determine whether we understand the control code or not. If not, we just set the status to something other than success and break out of the `switch` block.

The last piece of generic code we need is to complete the IRP after the `switch` block, whether it succeeded or not. Otherwise, the client will not get a completion response:

```
Irp->IoStatus.Status = status;
Irp->IoStatus.Information = 0;
IoCompleteRequest(Irp, IO_NO_INCREMENT);
return status;
```

We just complete the IRP with whatever status happens to be. If the control code was not recognized, that would be an failure status. Otherwise, it would depend on the actual work done in the case we do recognize the control code.

The last piece is the most interesting and important one: doing the actual work of changing a thread's priority. The first step is to check whether the buffer we received is large enough to contain a `ThreadData` object. The pointer to the user-provided input buffer is available in the `Type3InputBuffer` member and the input buffer length is at `InputBufferLength`:

```
if (stack->Parameters.DeviceIoControl.InputBufferLength < sizeof(ThreadData)) {
    status = STATUS_BUFFER_TOO_SMALL;
    break;
}
```

> You may be wondering, whether it's actually legal to access the provided buffer. Since this buffer is in user space, we must be in the context of the client's process. And indeed we are, as the caller is the client's thread itself that transitioned to kernel mode as described in chapter 1.

Next, we can assume the buffer is large enough, so let's treat it as a `ThreadData`:

```
auto data = (ThreadData*)stack->Parameters.DeviceIoControl.Type3InputBuffer;
```

If the pointer is `NULL`, then we should abort:

```
if (data == nullptr) {
    status = STATUS_INVALID_PARAMETER;
    break;
}
```

Next, let's see if the priority is in the legal range of 1 to 31, and abort if not:

```
if (data->Priority < 1 || data->Priority > 31) {
    status = STATUS_INVALID_PARAMETER;
    break;
}
```

We're getting closer to our goal. The API we would like to use is `KeSetPriorityThread`, prototyped as follows:

```
KPRIORITY KeSetPriorityThread(
    _Inout_ PKTHREAD Thread,
    _In_ KPRIORITY Priority);
```

The `KPRIORITY` type is just an 8-bit integer. The thread itself is identified by a pointer to a `KTHREAD` object. `KTHREAD` is one part of the way the kernel manages threads. It's completely undocumented, but the point here is that we have the thread ID from the client and need to somehow get a hold of a pointer to the real thread object in kernel space. The function that can look up a thread by its ID is aptly named `PsLookupThreadByThreadId`. To get its definition, we need to add another `#include`:

```
#include <ntifs.h>
```

Note that you must add this `#include` before *<ntddk.h>*, otherwise you'll get compilation errors.

Now we can turn our thread ID into a pointer:

```
PETHREAD Thread;
status = PsLookupThreadByThreadId(ULongToHandle(data->ThreadId), &Thread);
if (!NT_SUCCESS(status))
    break;
```

There are several important points in this code snippet:

- The lookup function is typed as accepting a `HANDLE` rather than some kind of ID. So is it a handle or an ID? It's an ID typed as handle. The reason has to do with the way process and thread IDs are generated. These are generated from a global private kernel handle table, so the handle "values" are actual IDs. The `ULongToHandle` macro provides the necessary casting to make the compiler happy. (Remember that a `HANDLE` is 64 bit on 64 bit systems, but the thread ID provided by the client is always 32 bit.)

- The resulting pointer is typed as PETHREAD or pointer to ETHREAD. Again, ETHREAD is completely undocumented. Regardless, we seem to have a problem since KeSetPriorityThread accepts a PKTHREAD rather than PETHREAD. It turns out these are the same, because the first member of an ETHREAD is a KTHREAD (the member is named Tcb). We'll prove all this in the next chapter when we use the kernel debugger. The bottom line is we can safely switch PKTHREAD for PETHREAD or vice versa when needed without a hitch.
- PsLookupThreadByThreadId can fail for a variety of reasons such as illegal thread ID or thread that has since terminated. If the call fails, we simply drop out of the switch with whatever status was returned from the function.

Now we are finally ready to change the priority. But wait - what if after the last call succeeds the thread is terminated, just before we set its new priority? Rest assured, this cannot actually happen. Technically, the thread can terminate at that point, but that will not make our pointer a dangling one. This is because the lookup function, if successful, increments the reference count on the kernel thread object, so it cannot die until we explicitly decrement the reference count. Here is the call to make the priority change:

```
KeSetPriorityThread((PKTHREAD)Thread, data->Priority);
```

All that's left to do now is decrement the thread object's reference; otherwise, we have a leak on our hands, which will only be resolved in the next system boot. The function that accomplishes this feat is ObDereferenceObject:

```
ObDereferenceObject(Thread);
```

And we're done! For reference, here is the complete IRP_MJ_DEVICE_CONTROL handler, with some minor cosmetic changes:

```
_Use_decl_annotations_
NTSTATUS PriorityBoosterDeviceControl(PDEVICE_OBJECT, PIRP Irp) {
    // get our IO_STACK_LOCATION
    auto stack = IoGetCurrentIrpStackLocation(Irp); // IO_STACK_LOCATION*
    auto status = STATUS_SUCCESS;

    switch (stack->Parameters.DeviceIoControl.IoControlCode) {
    case IOCTL_PRIORITY_BOOSTER_SET_PRIORITY: {
        // do the work
        auto len = stack->Parameters.DeviceIoControl.InputBufferLength;
        if (len < sizeof(ThreadData)) {
            status = STATUS_BUFFER_TOO_SMALL;
            break;
        }
```

```
        auto data = (ThreadData*)stack->Parameters.DeviceIoControl.Type3InputBuffer;
        if (data == nullptr) {
            status = STATUS_INVALID_PARAMETER;
            break;
        }

        if (data->Priority < 1 || data->Priority > 31) {
            status = STATUS_INVALID_PARAMETER;
            break;
        }

        PETHREAD Thread;
        status = PsLookupThreadByThreadId(ULongToHandle(data->ThreadId), &Thread);
        if (!NT_SUCCESS(status))
            break;

        KeSetPriorityThread((PKTHREAD)Thread, data->Priority);
        ObDereferenceObject(Thread);
        KdPrint(("Thread Priority change for %d to %d succeeded!\n",
            data->ThreadId, data->Priority));
        break;
    }

    default:
        status = STATUS_INVALID_DEVICE_REQUEST;
        break;
    }

    Irp->IoStatus.Status = status;
    Irp->IoStatus.Information = 0;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return status;
}
```

Installing and Testing

At this point we can build the driver and client successfully. Our next step is to install the driver and test its functionality. You can try the following on a virtual machine, or if you're feeling brave enough - on your dev machine.

(C)2019 Pavel Yosifovich

First, let's install the driver. Open an elevated command window and install using the *sc.exe* tool as we did back in chapter 2:

```
sc create booster type= kernel binPath= c:\Test\PriorityBooster.sys
```

make sure *binPath* includes the full path of the resulting SYS file. The name of the driver (*booster*) in the example is the name of the created registry key, and so must be unique. It doesn't have to be related to the SYS file name.

Now we can load the driver:

```
sc start booster
```

If all is well, the driver would have started successfully. To make sure, we can open *WinObj* and look for our device name and symbolic link. Figure 4-1 shows the symbolic link in *WinObj*.

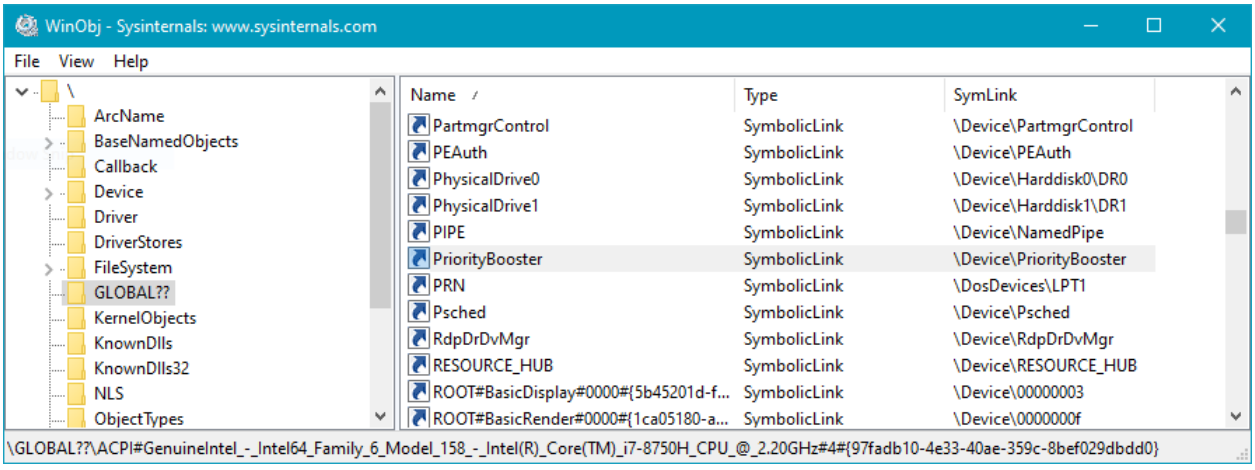

Figure 4-1: Symbolic Link in *WinObj*

Now we can finally run the client executable. Figure 4-2 shows a thread in *Process Explorer* of a *cmd.exe* process selected as an example for which we want set priority to a new value.

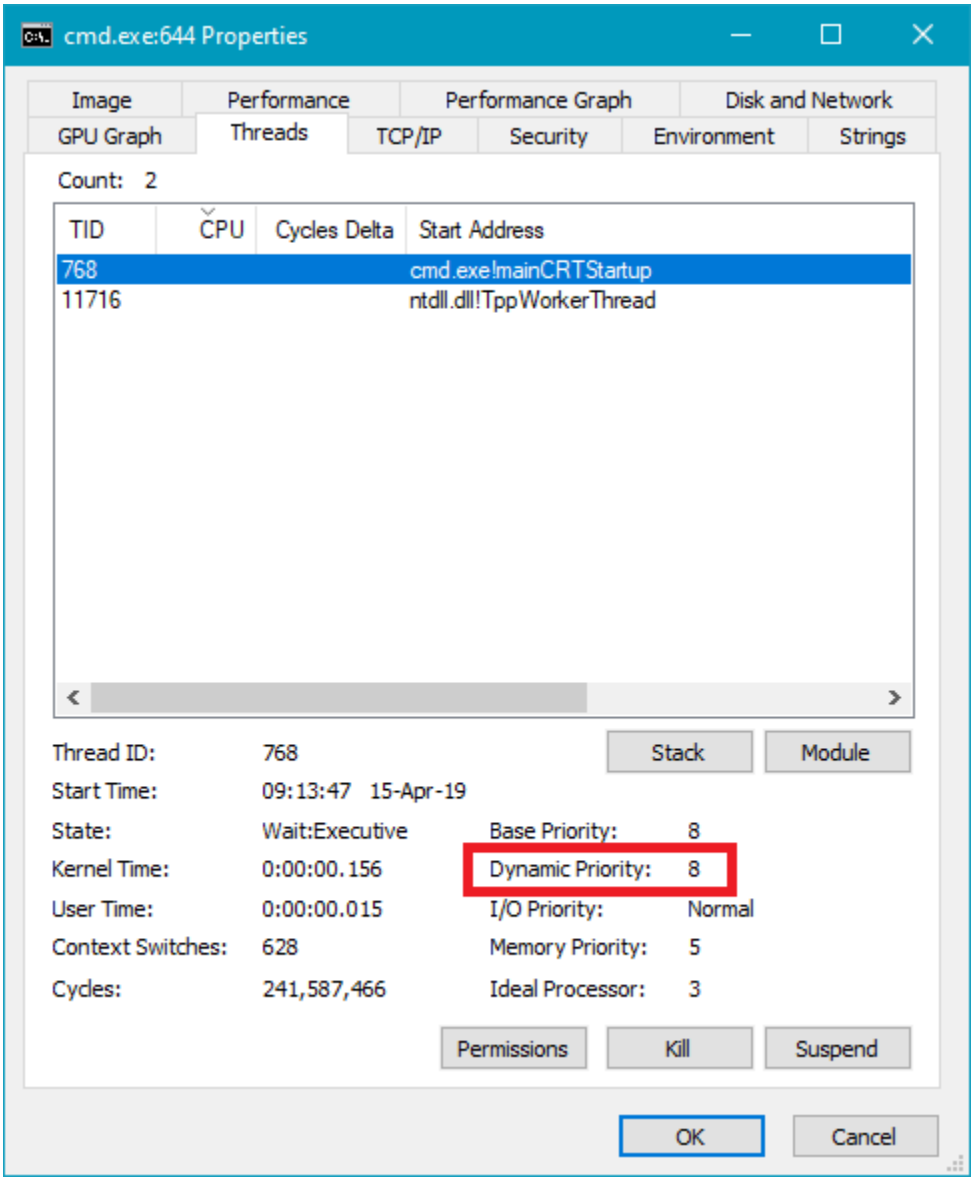

Figure 4-2: Original thread priority

Run the client with the thread ID and the desired priority (replace the thread ID as needed):

```
booster 768 25
```

If you get an error trying to run the executable, you may need to set the runtime library to a static one instead of a DLL. Go to Project properties, C++ node, **Code Generation**, and select **Multithreaded Debug**.

And voila! See figure 4-3.

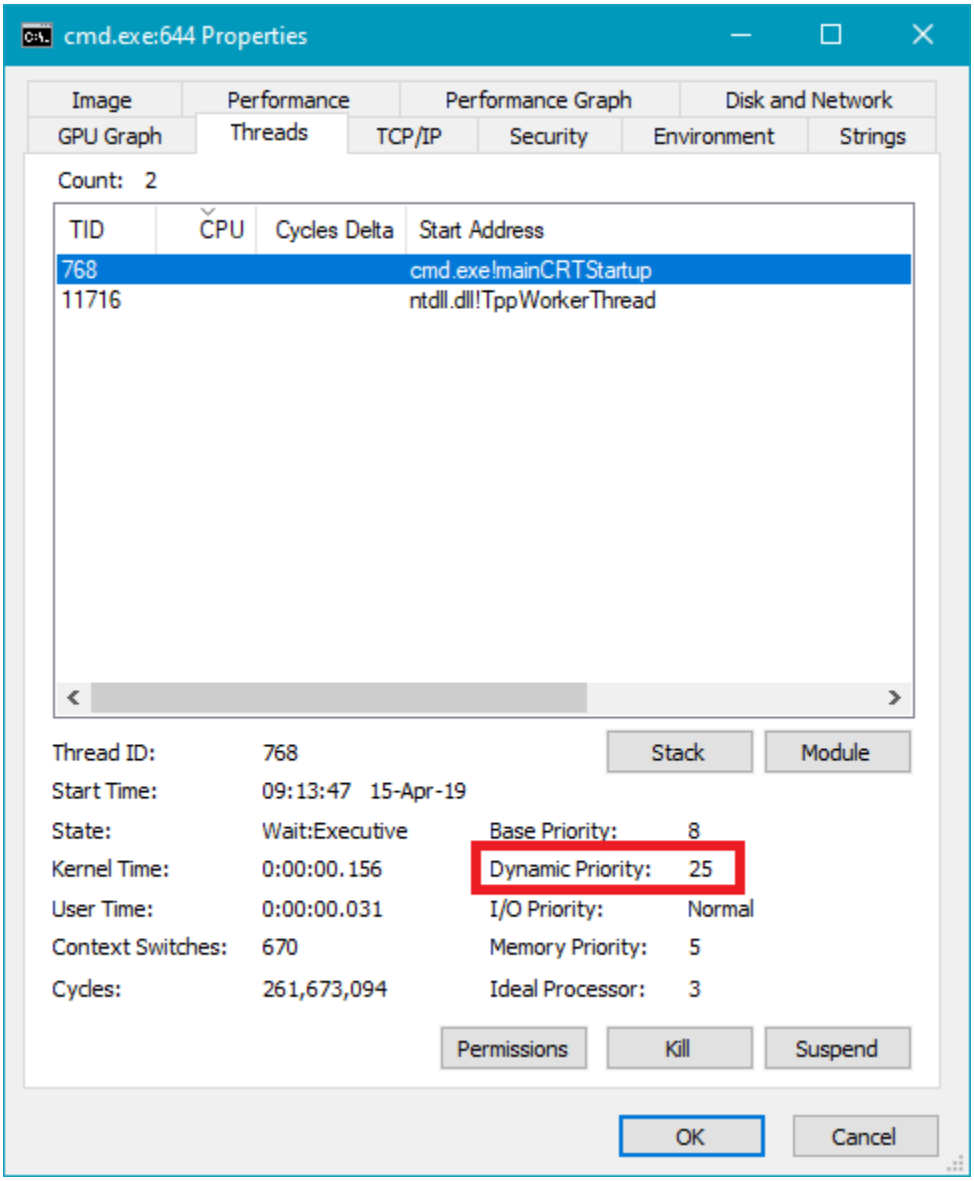

Figure 4-3: Modified thread priority

Summary

We've seen how to build a simple, yet complete, driver, from start to finish. We created a user mode client to communicate with the driver. In the next chapter, we'll tackle debugging, which is something we're bound to do when writing drivers that may not behave as we expect.

(C)2019 Pavel Yosifovich